Funktionsausdrücke

Funktionsdeklarationen = syntaktisch ein Statement
Funktionsausdrücke können anonym sein (Funktionen brauchen keinen Namen)

Funktionsdefinition (Anonyme Funktionen)

```
let tueDiesUndDas = function() {
  //Programmblock
};

tueDiesUndDas();
```

können auch Name haben, um sich innerhalb der Funktion selbst aufzurufen

um Funktionsausdruck ansprechen zu können, muss sie an Namen gebunden werden – mit let oder const - Name der Funktion sollte ihre Aufgabe klar herausstellen

um Funktion zu verwenden, braucht sie Funktionsaufruf
Funktionsaufruf besteht aus Bezeichner und runden Klammern

Funktionsaufruf bei Funktionsdeklaration auch von Position vor Funktion möglich

Ist bei Funktionsausdrücken nicht möglich – führt zu TypeError

Parameter-Übergabe

Parameter = spezieller Variablentyp, bei der Zuweisung anhand der Parameterübergabe stattfindet

im Hintergrund weist Funktion dem Parameter das übergebene Argument zu
kann dann innerhalb der Funktion genutzt werden

wenn kein Argument übergeben, Parameter = undefined

kann default-Wert setzen oder in Funktion prüfen

Unterschied zwischen Argument und Parameter nicht sonderlich bedeutsam

oft wird von Parametern geredet, auch wenn eigentlich Argument gemeint und umgekehrt

Funktion von Funktionen

schaffen Möglichkeit, wiederholt auf Code zuzugreifen

können Werte zurückzugeben

Funktion nimmt Parameter entgegen, verarbeitet sie und liefert Ergebnis der Verarbeitung zurück

jede Funktion = eigenes kleines Programm, das nach EVA-Prinzip (engl.: IPO-Model) arbeitet:

- Eingabe / Input
- Verarbeitung / Process
- Ausgabe / Output

Scopes (Gültigkeitsbereiche) von Funktionen

Variablen, die in Funktion definiert, können nicht außerhalb der Funktion erreicht werden, weil nur im Scope (Gültigkeitbereich) der Funktion definiert

Funktion kann alle Variablen und Funktionen erreichen, die im Scope definiert wurden, in dem auch Funktion definiert wurde
d.h. im globalen Scope definierte Funktion kann alle Variablen und Funktionen des globalen Scopes erreichen

wenn Funktion in Funktion definiert, kann innere Funktion auf alle Definitionen seiner Elternfunktion und alle Definitionen, auf die Elternfunktion Zugriff hat, zugreifen

```
let num1 = 20, num2 = 3, name = Chris;     // Variablen sind im globalen Scope definiert

function sum() {                           // Funktion ist im globalen Scope definiert

  return num1 + num2;
}

console.log(sum()); // gibt 23 zurück
```

Beispiel für verschachtelte Funktionen

```
function getScore() {
  let num1 = 2,  num2 = 3;

  function add() {
    return name + ' scored ' + (num1 + num2);
  }

  return add();
}

console.log(getScore()); // gibt "Chris scored 5" zurück
```

Guards - Wächter für Funktionen

return ermöglicht es auch, Funktion vorzeitig zu verlassen

ist aber Verletzung des Single Entrance/Single Exit-Prinzips (SESE)

SESE-Prinzip (in strukturierter Programmierung begründet) sagt, dass **jede Funktion genau einen Ein- und einen Aussprungspunkt** haben soll

sonst schwerer, Ablauf von Funktion zu verfolgen - entstehen oft subtile Bugs

begründete Ausnahme des Prinzips sind **guards** (Wächter)

guard oder guard clause am Funktions-Anfang prüft, ob Funktion mit übergebenen Argumenten Arbeit verrichten kann – wenn nicht, stoppt Wächter Funktions-Ausführung vorzeitig

IIFE

steht für Immediately-invoked Function Expression
ist sofort ausgeführter Funktionsausdruck

Beispiel: Grundmuster

```
(function () {
   // statements
})();
```

schafft Namensräume für mit var initialisierte Variablen, um verschiedene Code-Einheiten gegen andere abgrenzen zu können

besonders dann sinnvoll, wenn Projekt größer und komplexer wird oder man Bibliotheken erstellt, die neben denen anderer Hersteller eingesetzt werden sollen

Funktion erzeugt Scope – schafft privaten Raum für innen liegende Variablen und Funktionen
Deklaration innerhalb eines Scopes hat Vorrang vor außerhalb davon verwendeten
außerhalb deklarierte Variablen oder Funktionen gleichen Namens werden verdeckt

Die Deklaration einer Funktion – function foo() {} – ist kein ausführbarer Ausdruck. Erst der Aufruf mit dem Namen gefolgt von runden Klammern erzeugt den ausführbaren Ausdruck: foo().

Um eine anonyme Funktion direkt an Ort und Stelle auszuführen, muss der Parser erkennen, dass er einen Ausdruck vor sich hat und keine Deklaration einer Funktion. Eine Form der Notation sind die runden Klammern rund um die Funktion – in JS können Ausdrücke nicht in Klammern sitzen

IIFE mit Return-Wert

IIFEs können einen Namen haben und auch einen Return-Wert abliefern.

wenn an eine Variable gekoppelt, enthält Variable den Rückgabewert und nicht die Funktion

Verkürzte Schreibweise

Wenn man sich nicht um einen Return Value kümmern muss und es unrelevant ist, dass der Code etwas schwerer lesbar wird, kann man ein Byte sparen, indem man einen Unary Operator-Präfix vor die Funktion setzt.

```
!function(){ /* Anweisungen */ }();
```

Schaffung von Namensräumen für let-Variablen auch einfach über Block möglich

```
{…};
```

Beispiel: Ausdrücke und Funktionen, in Variablen und direkt verwendet

```
// Arithmetischer Ausdruck, in einer Variablen zwischengespeichert
let x = a + 2;
alert(x);

// Arithmetischer Ausdruck, sofort verwendet
alert(a + 2);

// Funktionsausdruck, zwischengespeichert und dann aufgerufen
let t = function(x) { return x + 7; };
alert (t(3));

// Funktionsausdruck, sofort verwendet
alert( (function(x) { return x + 7; })(3));
```

Arrays

Array = Liste von Elementen bzw. geordnete Sammlung von Daten
werden verwendet, um mehrere Werte in einziger Variablen zu speichern (Nicht-Array-Variable kann nur einen einzigen Wert speichern)

lassen sich erzeugen mithilfe von eckigen Klammern []

```
let hunde = new Array(); // erzeugt leeres Array

let hunde = new Array("Pinscher","Dackel","Hund","Boxer"); // erzeugt Array mit Werten
```

Kurzschreibweise:

```
let colors = [];  // erzeugt leeres Array

let meinArray = [1, 2, 3, 4];
```

Anzahl der Elemente im Array bestimmen:

```
console.log(meinArray.length); // => 4
```

Array-Zugriff mit Indexoperator

jedes Element in Array hat Nummer - auch numerischer Index genannt, über die man auf Element zugreifen kann (in JS beginnen Arrays mit Index 0)

Elemente innerhalb des Arrays stehen in Reihenfolge – wie nummerierte Liste
können anhand ihrer Position auf einzelne Elemente zugreifen

0 - Element 1
1 - Element 2
2 - Element 3

Nummerierung = Index und Namen = Werte des Arrays

Zugriff auf einzelnen Namen mit Indexoperator - die eckigen Klammern []
werden direkt hinter Array notiert
innerhalb der Klammern wird Index des gesuchten Elements angegeben

```
console.log(meinArray[2]); // => 3
```

Array-Manipulation

push()

Funktion push() für Aufnahme weiterer Elemente ins Array - Element anhängen

neues Element wird der Funktion push als Argument übergeben
im Array taucht es an letzter Position auf

pop()

pop() entfernt Element am Ende des Arrays
Funktion gibt entferntes Element zurück, so dass es weiterverwendet werden kann

unshift()

unshift() Methode fügt ein oder mehrere Elemente am Anfang eines Array hinzu
gibt neue Länge des Arrays zurück

shift()

Methode shift() entfernt erstes Element eines Arrays und gibt Element zurück
shift() verändert Länge des Arrays - ist length = 0, wird undefined zurückgegeben

splice()

pop() entfernt Sie immer nur letztes Element eines Arrays
splice() erlaubt, Elemente an beliebiger Position zu löschen

erstes Argument enthält Position ab der Elemente entfernt werden sollen
zweites Argument bestimmt Anzahl der zu löschenden Elemente (0-…)

weitere Argumente sind neu hinzuzufügende Elemente
kann an Index-Position gleichzeitig Löschen und Hinzufügen

Array-Methoden

sortieren mit sort()

in Grundvariante:
alle Elemente werden in Strings umgewandelt
sortiert Array anhand seines UTF-16-Codepoints – d.h. Zahlen werden auch als Strings verglichen
= lexiographischer Vergleich der Elemente anhand der UTF-16-Tabelle

im Aufbaukurs werden wir eigene Callback-Function schreiben, um hier einzugreifen

Strings - Arrays - Strings / split() und join()

split() zerlegt String anhand eines Separators in Array
join() konvertiert Array zu String – Seperator kann übergeben werden

slice() erstellt Kopien

slice() Methode schreibt Kopie von Teil des Arrays in neues Array-Objekt von begin bis end (end nicht enthalten) - originales Array wird nicht verändert
bei negativen Index kennzeichnet end Versatz vom Ende der Sequenz: slice(2, -1) extrahiert vom dritten bis vorletzten Element der Sequenz

indexOf() findet Position von Elementen

Funktion indexOf() - Einsatz um Position von Element zu finden
bei keinem Fund, gibt sie -1 zurück

concat() führt Arrays zusammen

Methode concat() führt zwei oder mehr Arrays zu einem zusammen
ändert keine existierenden Arrays, gibt stattdessen neues zurück

kann auch Array und neue Arraywerte zu neuem Array zusammenführen

includes() prüft Array

prüft, ob Element im Array vorkommt (ab ECMAScript 2016)
kann Startparameter als zweites Argument übergeben

Referenz

let a = ["a", "b", "c", "d"];

Funktion	Zweck	Beispielaufruf	a.join("")	Rückgabewert (Datentyp)
push	fügt Elemente am Ende des Arrays an	a.push("x", "y")	abcdxy	6 (number)
pop	entfernt das Element am Ende des Arrays	a.pop()	abc	d (string)
unshift	fügt Elemente am Anfang des Arrays hinzu	a.unshift("x","y")	xyabcd	6 (number)
shift	entfernt das Element am Anfang des Arrays	a.shift()	bcd	a (string)
slice	fertigt eine Kopie von einem Teil des Arrays an	a.slice(2, 4)	abcd	c,d (array)
splice	entfernt und/oder ergänzt Elemente im Array	a.splice(2, 2, "x")	abx	c,d (array)
sort	sortiert das Array	a.sort()	abcd	a,b,c,d (array)
reverse	kehrt die Reihenfolge der Elemente um	a.reverse()	dcba	d,c,b,a (array)
concat	Verbindet ein Array mit einem oder mehreren Elementen oder Arrays	a.concat(["x","y"])	abcd	a,b,c,d,x,y (array)
includes (*)	Prüft, ob ein Element im Array vorkommt	a.includes("x"[,startindex])	abcd	false (boolean)
indexOf	Ermittelt die Position (index) eines Elements im Array	a.indexOf("c"[,startindex])	abcd	2 (number)
lastIndexOf	Ermittelt die Position (index) eines Elements im Array (vom Ende her)	a.lastIndexOf("c"[,startindex])	abcd	2 (number)
join	Verbindet alle Elemente eines Arrays zu einem String	a.join("==")	abcd	a==b==c==d (string)
toString	wie join, verwendet aber das Komma als festen Separator	a.toString()	abcd	a,b,c,d (string)

* ab ECMAScript 2016

Higher Order-Functions

Das Verhalten von sort() beeinflussen

in JS sind Funktionen einfach Werte

let price = 3;

let add = (a, b) => a + b;

Antwort auf Frage nach Typ der Variable:

typeof price; // => number

typeof add; // => function

funktionale Programmierung spricht von first class functions (meint, dass Sprache Funktionen genauso behandelt wie andere Werte)
kann Funktion mit Variable referenzieren und sie sogar als Argumente übergeben

[16, 10, 2, 12, 1].sort(); // => [1, 10, 12, 16, 2] lexikographische Sortierung…

müssen Funktion mitteilen, welcher von zwei Werten kleiner und welcher größer ist

benötigen dazu Funktion, die zwei Parameter entgegennimmt und miteinander vergleicht

sort() erwartet von Vergleichsfunktion, dass sie:

- 0 zurückgibt, falls a = b (es muss nichts geschehen)
- positive Zahl zurückgibt, falls a > b (es muss sortiert werden)
- negative Zahl zurückgibt, falls a < b (es muss nichts geschehen)

dazu einfach b von a subtrahieren:

"use strict";

let array = [16, 10, 2, 12, 1]

let zahlenPruefen = (a, b) => a - b;

array.sort(zahlenPruefen);

console.log(array); // => [1, 2, 10, 12, 16]

könnte Funktion auch direkt definieren, ohne sie vorher der Variable zuzuweisen

[16, 10, 2, 12, 1].sort((a, b) => a - b); // => [1, 2, 10, 12, 16]

oft Code besser lesbar, wenn Funktion vorher einer Variablen zugewiesen

durch Variablennamen zahlenPruefen kann Wartungsprogrammierer ursprüngliche Absicht (numerischer Vergleich) leichter nachvollziehen